Sample Midterm Answers

Question #1.

```
(define do-nothing
    (lambda () '(I am a function with no arguments)))
```

Question #2. Nothing interesting will happen; Scheme doesn't take a look at anything in the body of a function until we actually try to evaluate that function. Only until we ask Scheme something like `(problem-two 42)` will Scheme examine the body and suddenly realize that it has no clue what `muhahaha` is.

Question #3. Here is one possible answer that uses the `keep` higher order function. The idea is to filter out all sentence elements that fulfill the predicate, and compare that list against our original list.

```
(define (its-all-good? function sent)
  (equal? (keep function sent) sent))
```

In class, we compared lengths of those two lists, which also works:

```
(define (its-all-good? function sent)
  (equal? (count (keep function sent)) (count sent)))
```

but the first approach is more direct. It's a good idea to understand why both approaches will work.

Question #4. Let's be comprehensive for this problem, so that we get the full range of tools that Scheme lets us use. (This means that this answer will be somewhat long-winded. Apologies.)

```
(define (vowel? letter)
  (if (equal? letter 'a) #t
      (if (equal? letter 'e) #t
          (if (equal? letter 'i) #t
              (if (equal? letter 'o) #t
                  (equal? letter 'u))))))
```

This immediately shows the advantage of using the `cond` form; it's well suited for this kind of case analysis.

```
(define (vowel? letter)
  (cond ((equal? letter 'a) #t)
        ((equal? letter 'e) #t)
        ((equal? letter 'i) #t)
        ((equal? letter 'o) #t)
        (else (equal? letter 'u))))
```

or

```
(define (vowel? letter)
  (cond ((equal? letter 'a) #t)
        ((equal? letter 'e) #t)
        ((equal? letter 'i) #t)
        ((equal? letter 'o) #t)
        ((equal? letter 'u) #t)
        (else #f)))
```

Whew. Still long winded. Using `member?` is a much nicer looking definition:

```
(define (vowel? letter)
  (member? letter '(a e i o u)))


(define (vowel? letter)
  (member? letter 'aeiou))
```

will both work. Finally, here's one more version of `vowel?`:

```
(define (vowel? letter)
  (or (equal? letter 'a)
      (equal? letter 'e)
      (equal? letter 'i)
      (equal? letter 'o)
      (equal? letter 'u)))
```

Ok, that's enough.

Question #5. One possible answer to this would be to use a helper function:

```
(define (remove-vowel-from-word wd)
  (keep (lambda (letter) (not (vowel? letter))) wd))
```

which allows us to define `de-voweler` really easily:

```
(define (de-voweler sent)
   (every remove-vowel-from-word sent))
```

λ

For the sadistic λ lovers among you, we can squeeze everything into a single, gratuitous, hideous lambda expression:

```
(define de-voweler
  (lambda (sent)
    (every (lambda (wd)
             (keep (lambda (ch) (not (member? ch 'aeiou))) wd))
           sent)))
```

which only proves the point: even if we *could* do something like this, we shouldn't! In written English, we use paragraphs to break down an idea into digestible parts, and we use helper functions for the same reason in Scheme. It's simply indecent to write excessively lambdified expressions, so follow Occham's Razor whenever you can and try the straightforward route first.

Good luck to you all!